Research Inventy: International Journal Of Engineering And Science
Issn: 2278-4721, Vol. 2, Issue 7 (March 2013), Pp 1-7
Www.Researchinventy.Com

Graphics Processor Unit Hardware Acceleration of Levenberg-Marquardt Artificial Neural Network Training

[1] David Scanlan, [1] David Mulvaney
[1](*School of Electronic, Electrical and Systems Engineering, Loughborough University LE11 3TU, UK)*

Abstract *- This paper makes two principal contributions. The first is that there appears to be no previous a description in the research literature of an artificial neural network implementation on a graphics processor unit (GPU) that uses the Levenberg-Marquardt (LM) training method. The second is an initial attempt at determining when it is computationally beneficial to exploit a GPU's parallel nature in preference to the traditional implementation on a central processing unit (CPU). The paper describes the approach taken to successfully implement the LM method, discusses the advantages of this approach for GPU implementation and presents results that compare GPU and CPU performance on two test data sets*

Keywords *- Artificial Neural Networks, Graphics Processor Unit, Levenberg-Marquardt Networks*

I. INTRODUCTION

All desktop computers contain some form of graphics processing unit (GPU) and it is becoming increasingly common for manufacturers to provide the user with access to its programmable operations. The inherent parallelism, high data bandwidth and favourable cost to performance ratio that are features of modern GPUs, have made them an attractive choice for the computational acceleration of many applications, including fluid flow simulation [1], finite-element simulation [2] and ice crystal growth [3].

Neural networks' ease of use and semi-automatic adaption has made them a very desirable option for many applications, such as handwriting identification [4] and speech recognition [5]. A significant drawback is long calculation time, as, not only does the realisation of artificial neural networks (ANNs) often require the application of training data over many epochs, but also the repeated application of that data with different training parameters is needed to generate a solution with good classification performance. A number of alternative solutions have been developed to reduce this computational overhead, such as automatically tuning parameters to reduce the number of alternative ANN solutions that need be generated [6], novel training approaches that require fewer epochs [7] or accelerating the computations by using bespoke electronic hardware [8]. As this third approach is taken in this paper, it is important to mention that previous researchers looking to hardware acceleration have investigated a number of novel approaches. These include central processing units (CPUs) tailored to perform the calculations needed during training [9], mesh connected machines of architecture similar to that of interconnected neurons [10], or GPUs adapted to mirror the parallel nature of neural calculations [11].

GPUs have been previously used by a number of researchers to accelerate ANN classification, for example [12], [13] and [14]. In the literature, no previous GPU solution using the Levenberg-Marquardt (LM) training method has been described. This is probably due the fact that its calculation involves a matrix inversion operation that is appears to be computationally expensive even for parallel solution. This paper has adopted a solution for the matrix inversion operation that allows the LM algorithm to be implemented efficiently on a GPU. Note that a commercial LM solution exists for which the operational details have not been published, but for which it is claimed that a calculation speed improvement of up to 65 times can be obtained by choosing the GPU rather than the CPU implementation [15]. For the examples in this paper, the time taken to train the network using the GPU is shown to be up to ten times faster than a similar implementation run solely on the machine's CPU. In practice, the measured difference in performance will depend significantly on the specific GPU and CPU used in the comparison and these should always be specified alongside the quoted figures.

This paper briefly describes the LM algorithm, the general architecture of modern GPUs and the implementation of the ANN on the selected GPU. Finally, results are presented to compare the training times of the GPU and CPU on two test data sets.

II. LEVENBERG-MARQUARDT ARTIFICIAL NEURAL NETWORKS

In an ANN, each neuron will typically apply an activation function to a weighted sum of inputs and provide a single output. During supervised learning, a set of training vectors with known outputs is repeatedly applied over a number of epochs and the weight values are altered in such a way as to improve the overall classification performance of the ANN. Such a training process can be performed by one of a number of algorithms, the most popular being backpropagation [16], but LM [17] and Conjugate Gradient Descent [18] are also in common use. Unsupervised learning methods are also available, but these are beyond the scope of this paper. When the weights are updated after presenting each input vector to the network this is known as online training; the alternative being batch training where all the training vectors are applied before the weights are updated. There is also a hybrid of the two which uses mini-batches of the total data set before applying the weight update

The neurons themselves can be interconnected in a number of ways, but, due to its simplicity and regular structure, the most commonly used architecture is the multi-layer perceptron (MLP) feed-forward network. An example MLP network with 11 input neurons, four hidden neurons and two output neurons is shown in Fig. 1. In the general case, for M input neurons i_m, P hidden neurons h_p and one output neuron o, the weights on the edges between the input and hidden layers can be represented by W_{pm} and those between the hidden and output layer (assuming a single output neuron) by w_p. Given k input vectors, input value m is given the value i_m^{γ} when presented with vector γ where $\gamma=\{1,2,\ldots,k\}$.

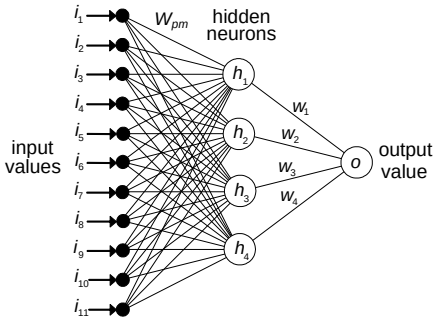


Fig. 1. Example of an MLP ANN with a single output

The LM algorithm has recently become increasingly popular as its second-order optimisation techniques allow a very efficient batch update method. A drawback of the LM approach is that the ANN must have only a single output, but this can be overcome by implementing multiple networks [19]. For the detailed mathematics underlying the LM algorithm, the reader is referred to Marquardt [20], but general LM algorithm is shown in Fig. 2 and is briefly explained below.

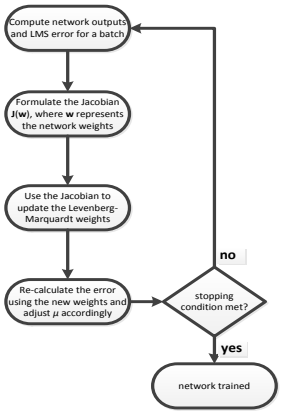


Fig. 2. Flow diagram outlining the procedure for using the Levenberg- Marquardt training algorithm

Each batch of data is fed forward through the network, as described previously, to obtain a vector of output values each calculated using equation (1) below, where z is the activation function.

$$o^{\gamma} = z\left(\sum_{p=1}^{P} z\left(\sum_{m=1}^{M} i_m^{\gamma} W_{pm}\right) w_p\right) \qquad (1)$$

The least mean-square (LMS) error can then be obtained using

$$E[w] = \frac{1}{2}\sum_{\gamma=1}^{k} (R^{\gamma} - o^{\gamma})^2 \ , \qquad (2)$$

where R^{γ} is the desired output from the ANN for a specific input vector γ. The Jacobian matrix used in LM requires a vector of all the weights contained within the network to calculate a matrix of partial derivatives (with respect to each weight individually) for each input pattern in the batch. The Jacobian is given by

$$\mathbf{J}(\mathbf{w}) = \begin{pmatrix} \frac{\partial e_1(\mathbf{w})}{\partial w_1} & \cdots & \cdots & \frac{\partial e_1(\mathbf{w})}{\partial w_v} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{\gamma}(\mathbf{w})}{\partial w_1} & \cdots & \cdots & \frac{\partial e_{\gamma}(\mathbf{w})}{\partial w_v} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_p(\mathbf{w})}{\partial w_1} & \cdots & \cdots & \frac{\partial e_p(\mathbf{w})}{\partial w_v} \end{pmatrix} , \qquad (3)$$

where $v = MP + 2P$ and $\mathbf{w}$ is a vector of weights $\mathbf{w} = [W_{11},..,W_{PM}, B_1,..,B_P, w_1,..,w_P]^{\mathrm{T}}$, where the B_p values are the bias values of the hidden neurons. To update the weights during training the LM algorithm determines a weight update vector $\mathbf{\Delta w}$, calculated by

$$\Delta\mathbf{w} = \left[\mathbf{J}^{\mathrm{T}}(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu\mathbf{I}\right]^{-1}\mathbf{J}^{\mathbf{T}}(\mathbf{w})\mathbf{e} \ , \qquad (4)$$

where $\mathbf{e}$ is the vector containing errors for each input vector in the batch and $\mathbf{I}$ is the identity matrix of dimension v. The new weights can now be calculated by

$$\mathbf{w_{new}} = \mathbf{w_{old}} + \Delta\mathbf{w} \ . \qquad (5)$$

The ANN's LMS error with new weights is now computed. If the new error is smaller than the previous error then μ is reduced by a factor μ^-, but if the error is larger than the previous error u is increased by a factor of μ^+. The values of μ, μ^- and μ^+ are all training parameters that must be selected before training the network.

III. GRAPHICS PROCESSOR UNIT IMPLEMENTATION

A CPU's pipeline is instruction flow-driven, whereas a GPU uses a data-dependent control flow that is better suited to construct and output graphics to a monitor. In GPUs, all data are represented as a stream of a common data type that passes through the pipeline as a single entity with each element in the stream being operated upon in an identical manner. The main components that a stream will pass through in the GPU are the vertex processor, the fragment processor and the memory system.The vertex processor receives vertices from an application and operates on these primitives to generate a screen position, a colour and a texture coordinate. Fixed function hardware operations are then applied such as clip and cull. In the fragment processor, each texture element or *texel* (similar to a single pixel displayed on the screen) is processed on by a shader program. A texel is made up of four floating point components, namely red, green, blue and alpha (opacity). After leaving the fragment processor, a texel passes through some tests, such as a depth test (or z-cull), as well as other fixed procedures, such as alpha blending. Both the vertex processor and the fragment processor are programmable in most modern GPUs, giving the programmer considerable flexibility when deploying a graphics application.The type of memory, bus width and memory clock frequency used in a GPU, as well as the interface the GPU has with the system memory, determine the speed at which data can be transferred between the two. Most graphics cards ultilize a form of double-data rate (DDR) memory such as Graphics DDR (GDDR)

providing data bandwidths above 3 Gbit/s. The transfers between GPU and CPU are determined by the graphics card interface bus employed, with the current standard, PCI Express 2.0, having a maximum bandwidth of 8 GB/s

The majority of graphics programs are developed to communicate with either SGI's OpenGL [21] or Microsoft's Direct3D [22] drivers. Alternatively, at a lower level somewhat akin to assembler languages, programs known as 'shaders' can be loaded and run on each of the vertex and fragment programmable processors.In the current work, the Brook GPU program language was used [23]. Brook extends the functionality of the C or C++ language, allowing extra data types that define structures of floating point numbers to match the native architecture of GPUs. For example, the *float4* data type is simply a structure of four floating values that matches the texel representation. By creating a standard texture on the GPU it is possible to create a 2D array of float4 structures. The major advantage of such an implementation is that the vector pipeline will be able to operate on each component independently. In contrast, if an array of single floats was required, then, when mapped to a single stream, only 25% of the fragment processor pipeline would be utilised. Hence, mapping data to make best use of a GPUs parallel processing capabilities is important in achieving the best acceleration of a given application.

A further consideration in making best use of a GPUs capabilities is the quantity of data that needs to be transferred between the computer's main memory and the GPU's local memory. In general, applications that adapt well to GPU implementation are those that are computationally intensive yet can principally operate on data that resides in local memory. Sharing data with the rest of the computer system requires their transfer between the GPU's streams incurring time penalties which, if they accumulate, would be detrimental to performance. In any given application, data will need to be shared with the CPU and, to achieve acceleration, the improvement in performance that results from moving stream operations to the GPU should outweigh the overheads associated with the transfers involved.

IV. NEURAL NETWORK IMPLEMENTATION

The GPU used to generate the results in this paper is an NVIDIA GeForce 6800 GS with GDDR3 memory, operating on the PCI-Express X16 bus, includes 512MB of memory and provides a memory bus width of 256 bits. The CPU used for comparison purposes is an AMD Athlon64 3.5GHz with 1GB of DDR system memory.

4.1 Design Overview

The implementation described in this paper concentrates on multiple-input single-output networks with a single hidden layer as this architecture meets the requirements of the LM algorithm. Although the ANN is limited to one hidden layer, the number of neurons in this layer can be chosen by the user. In order to make good use of the GPU's parallel pipelines, up four networks are created at a time and trained concurrently. For networks with more than four outputs, the process is repeated once the first four have been calculated. Fig. 4 shows how a network with multiple outputs is treated as multiple networks with single outputs to utilise the GPU's vector pipeline to best effect.

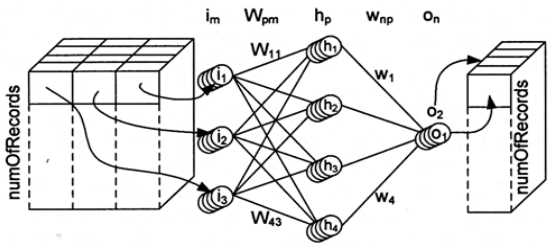


Fig. 3. Multiple single-output neural networks running in parallel on the GPU, using all four components of the float4 data type.

The inputs are replicated into each of the streams' four components and the outputs split into groups of four and assigned one channel each. The weights are randomly assigned values in the range $\left[-1/\sqrt{N}, 1/\sqrt{N}\right]$, where N is the total number of inputs of the neuron that is connected to the weight's output. This method of weight initialisation has been shown in [24] to lead to better convergence than initialising the weights randomly in the range [-0.5,0.5].

Batch training methods are better suited to GPU implementation than single pattern online methods as they operate on a single large data set for each calculation. This means that the data set can be loaded into the GPU memory for training rather than loading each training vector individually on each cycle through the loop. The LM algorithm is a batch training method known to outperform most other training methods in terms of calculation time for medium-sized ANN with a few hundred weights or more, but has been little used in GPU implementations as it requires a matrix inversion operation that appears to be not well supported by the GPU architecture. However, it has been shown by Galoppo *et al.* [25], that algorithms such as Gaussian Elimination and LU decomposition (where L and U refer to the lower and upper triangular matrices) that are able to calculate a matrix inverse can be adapted to run efficiently on a parallel GPU implementation.

4.2 Software

To compare the time taken to train the neural network on the GPU, two versions of the program were required. One used solely CPU code and the second principally using the streams on the GPU. For the GPU version, initial data such as the input patterns, weights and known outputs are first loaded into CPU memory and then transferred into streams. It should be noted that this costly data transfer is only performed once per training of a set of four networks. The program utilises the float4 data type throughout and is able to train four single output networks simultaneously, keeping all required data in streams in the fast on-chip memory. Only once the set of networks has been trained are the final results and weights read back from the GPU to the system memory to be saved. Once a network has been trained for each output, a set of test data may be used to verify the success of the training. The test data is provided in a similar method to the training data and a log file produced detailing the results. Using the saved weights for all the networks allows them to be used collectively for predicting the output of any input test vector.

4.3 Limitations

The size of datasets used in any part of the calculation or training of the neural network is strictly limited to the texture size available from the GPU. In the case of the 6800GS this is l024xl024. In most cases this is quite adequate but if a larger batch size was required, a hybrid semi-batch method could be used and implemented in the software. The LM algorithm requires an approximation to the 2D Hessian matrix of dimension *v* x *v*. As there is only a single output per network this effectively means that ANNs with large numbers of hidden and inputs neurons cannot be supported. The need to train four single-output networks in parallel does restrict the stopping criterion options that can be made available. If the user requires a stopping condition that relates to the output error of the network, the training of all the streams of ANNs would need to continue until all four networks had reached the desired error, effectively wasting computational effort.

V. RESULTS

The timing results were obtained using relevant routines available in the Brook compiler and the OpenGL libraries to provide performance counters with millisecond accuracy.Two data sets obtained from ultrasonic reflections from obstacles were used to generate the results; further information on the training data can be found in [26]. Both data sets had 26 continuous inputs and 11 discrete outputs, with the first data set contained 107 training vectors and the second 2562 vectors. Stop sets and test sets were also available. To demonstrate the importance of the numbers of neurons in the networks in this study, a single ANN with four inputs and a single output was trained on the first data set to classify a single output class only and the timing results are shown in Fig. 4. The CPU outperforms the GPU not only because together the network and training data are small enough to fit in the CPU's cache and can therefore be accessed quickly, but also the single output means the GPU is essentially working at only a quarter of its potential throughput capacity.

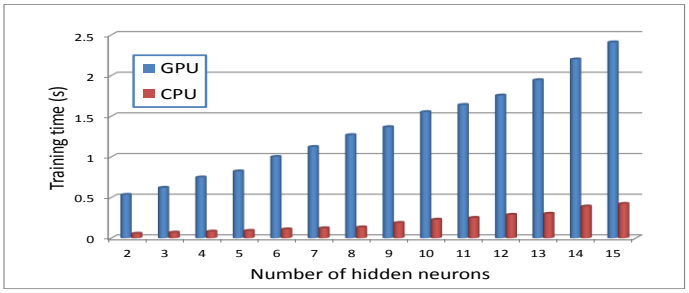
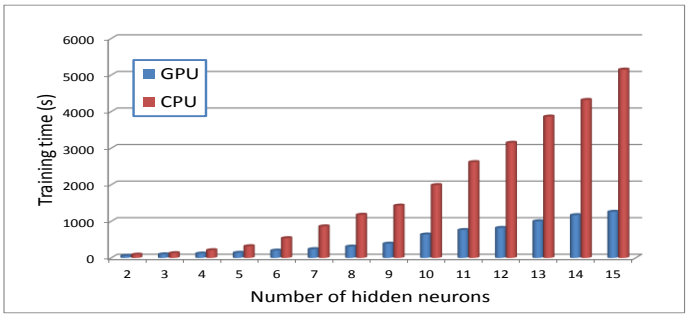
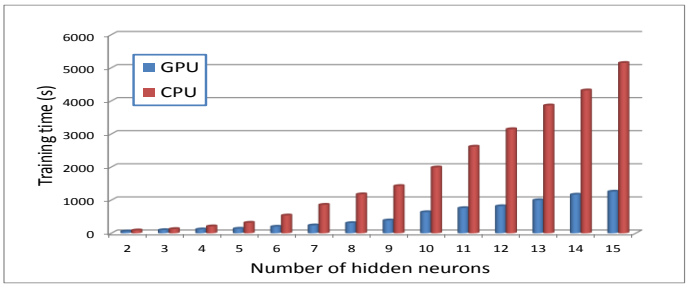


Fig. 4. Relative times taken to train a single ANN using the ultrasonic data with different numbers of hidden neurons.

For the second data set, 11 separate neural networks were trained, one for each output class. The GPU now outperforms the CPU, as shown in Fig. 5. This is in spite of the fact that, due to the texture size limitation, only a subset of these vectors could be applied in a single batch and consequently copying of data from the main computer memory to the GPU's on-chip memory was needed during training.



(a) batch size of 512 data vectors



(b) batch size of 1024 data vectors

Fig. 5. Times taken to train a set of 11 ANNs using the ultrasonic data with different numbers of hidden neurons.

The results show that there is an increase in time required to train larger networks and, in general, this may either result from including more hidden neurons (as here) or be due to the fact that a larger batch size is available. For other larger ANNs tested, the GPU training was found to outperform the CPU by a factor of between three and ten.

VI. CONCLUSION

The consumer demand for increasingly powerful graphic performance in applications such as high-definition television and gaming has led to substantial improvements in the computational power of GPUs, leading to a continual fall in the cost-performance ratio. Many scientific applications have already begun to use the GPU as a coprocessor for computationally intensive work. However, GPUs are by no means a suitable platform for all applications, as the reconfiguration of an application requires that careful thought be given to many aspects of the design, including data-driven control flow, identification of appropriate parallelisms in the code and efficient memory usage in data storage. The accessibility of GPU hardware is continually improving and both NVIDIA [27] and ATI [28] have not only made development tools available at a reasonable cost, but have also produced excellent supporting documents to ease the learning process.

The work described in this paper has shown the methods and concepts that are required to map ANN using the LM training algorithm to a GPU. A number of modifications could be made to the network architecture, such as the implementation of multiple layers of ANNs [19]. Although the LM training method is notorious for its long computation time, its overall training time and classification performance are generally favourable in comparison with other approaches. As the size of a batch is restricted by the texture size offered by the GPU, a modification that allows the software to operate in a semi-batch mode could allow larger training datasets to be used. Very large ANNs, trained using large data sets that need to be streamed to the GPU would yield the greatest benefits in terms of reduced calculation time. The further goal could be the development of a distributed system, essentially creating a GPU cluster.

REFERENCES

[1] Z. Fan, F. Qui, A. Kaufman and S. Yoakum-Stover, GPU cluster for high performance computing, Proc. ACM/IEEE Conf. On Supercomputing, Pittsburgh, PA, 2004, 47-58

[2] M. Rumpf and R Strzodka, Using graphics cards for quantized FEM computations, Proc. Visualization, Imaging and Image Processing Conf., Marbella, Spain, 2001, 193-202.

[3] T. Kim and M. Lin, Visual simulation of ice crystal growth, Proc. ACM SIGGRAPH Eurographics Symp. on Computer Animation, San Diego, CA, 2003, 86-97.

[4] I.-S. Oh and C.Y. Suen, Distance features for neural network-based recognition of handwritten characters, Int. J. of Document Analysis and Recognition, 1(2), 1998, 73-88.

[5] E. Trentin, M. Gori, Robust combination of neural networks and hidden Markov models for speech recognition, IEEE Trans. Neural Networks, 14(6), 2003, 1519-1531.

[6] L. Behera, S. Kumar and A. Patnaik, On adaptive learning rate that guarantees convergence in feedforward networks, IEEE Trans. Neural Networks, 17(5), 2006, 1116-1125.

[7] X. Liang, Removal of hidden neurons in multilayer perceptrons by orthogonal projection and weight crosswise propagation, Neural Computing and Applications, 16(1), 2007, 57-68.

[8] J Misra and I Saha, Artificial neural networks in hardware: A survey of two decades of progress. Neurocomputing, 74(1–3), 2010, 239-255.

[9] R. F. Lyon and L. S. Yaeger, On-line hand-printing recognition with neural networks, Proc. 5th Int. Conf. on Microelectronics for Neural Networks and Fuzzy Systems, Torino, Italy, 1996, 201-212.

[10] R. A. Ayoubi and M. A. Bayoumi, Efficient mapping algorithm of multilayer neural network on Torus architecture, IEEE Trans. Parallel and Distributed Systems, 14(9), 2003, 932-943.

[11] K. Oh and K Jung, GPU implementation of neural networks, Pattern Recognition, 37(6), 2004, 1311-1314.

[12] R. Dolan and G. DeSouza, GPU-based simulation of cellular neural networks for image processing, Proc. 2009 Int. Joint Conf. on Neural Networks, Atlanta, GA, 2009, 2712-2717.

[13] H. Jang, A. Park, K. Jung, Neural network implementation using CUDA and OpenMP, Proc. Digital Image Computing: Techniques and Applications, Canberra, Australia, 2008, 155-161.

[14] T-Y Hoa, P-M Lama and C-S Leung, Parallelization of cellular neural networks on GPU, Pattern Recognition, 41(8), 2008, 2684-2692.

[15] Neurosolutions, http://www.neurosolutions.com/products/cuda, accessed 17 December 2012.

[16] J. Hertz, A. Krogh, R.G.Palmer, Introduction to the theory of neural computation, (Reading, MA: Addison-Wesley, 1991), 115-120.

[17] M. T. Hagan. and M. Menhaj, Training feed-forward networks with the Marquardt algorithm, IEEE Trans. Neural Networks, 5(6), 1994, 989-993.

[18] C. Charalambous, Conjugate gradient algorithm for efficient training of artificial neural networks, IEE Proc. G Circuits, Devices and Systems, 139(3), 1992, 301-310.

[19] D. J. Mulvaney and I. P. W. Sillitoe, The classification of ultrasonic signals using novel neural network approaches, Int. J. Robotics and Automation, 14(1), 1999. 15-22.

[20] D. W. Marquardt, An algorithm for least-squares estimation of nonlinear parameters, J. Soc. Industrial and Applied Mathematics, 11(2), 1963, 431-441.

[21] OpenGL, http://www.sgi.com/products/software/opengl, accessed 17 December 2012.

[22] DirectX, www.microsoft.com/en-us/download/details.aspx?id=35, accessed 17 December 2012.

[23] Brook GPU, http://graphics.stanford.edu/projects/brookgpu/index.html, accessed 17 December 2012.

[24] G. Thimm and E. Fiesler, High-order and multilayer perceptron initialization, IEEE Trans. Neural Networks, 8(2), 1997, 349-359.

[25] N. Galoppo, N. K. Govindaraju, M. Henson and D. Manocha, LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, Proc. ACM/IEEE Conf. On Supercomputing, Seattle, WA, 2005, 3.

[26] I.P.W. Sillitoe, L. C. Axbrink and D. J. Mulvaney, Extensible sonar classifiers for local environment mapping, Proc. Int. Conf. Applied Informatics, Innsbruck, Austria, 2001, 149-154.

[27] NVIDIA CUDA, https://developer.nvidia.com/category/zone/cuda-zone, accessed 17 December 2012.

[28] ATI development tools, http://developer.amd.com/tools/heterogeneous-computing/, accessed 17 December 2012.